Information and Software Technology 67 (2015) 237–253

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study

Antonio Martini *, Jan Bosch, Michel Chaudron

Computer Science and Engineering, Software Engineering, Chalmers University of Technology | Gothenburg University, Göteborg, Sweden

ARTICLE INFO

Article history:
Received 30 November 2014
Received in revised form 8 July 2015
Accepted 10 July 2015
Available online 17 July 2015

Keywords:
Architectural Technical Debt
Software management
Software architecture
Agile software development
Software life-cycle
Qualitative model

ABSTRACT

Context: A known problem in large software companies is to balance the prioritization of short-term with long-term feature delivery speed. Specifically, Architecture Technical Debt is regarded as sub-optimal architectural solutions taken to deliver fast that might hinder future feature development, which, in turn, would hinder agility.
Objective: This paper aims at improving software management by shedding light on the current factors responsible for the accumulation of Architectural Technical Debt and to understand how it evolves over time.
Method: We conducted an exploratory multiple-case embedded case study in 7 sites at 5 large companies. We evaluated the results with additional cross-company interviews and an in-depth, company-specific case study in which we initially evaluate factors and models.
Results: We compiled a taxonomy of the factors and their influence in the accumulation of Architectural Technical Debt, and we provide two qualitative models of how the debt is accumulated and refactored over time in the studied companies. We also list a set of exploratory propositions on possible refactoring strategies that can be useful as insights for practitioners and as hypotheses for further research.
Conclusion: Several factors cause constant and unavoidable accumulation of Architecture Technical Debt, which leads to development crises. Refactorings are often overlooked in prioritization and they are often triggered by development crises, in a reactive fashion. Some of the factors are manageable, while others are external to the companies. ATD needs to be made visible, in order to postpone the crises according to the strategic goals of the companies. There is a need for practices and automated tools to proactively manage ATD.

1. Introduction

Large software industries strive to make their development processes fast and more responsive, minimizing the time between the identification of a customer need and the delivery of a solution. The trend in the last decade has been the employment of Agile Software Development (ASD) [1]. At the same time, the responsiveness in the short-term deliveries should not lead to less responsiveness in the long run. To illustrate such a phenomenon, a financial metaphor has been coined, which relates taking sub-optimal decisions in order to meet short-term goals to taking a financial debt, which has to be repaid with interests in the long term. Such a concept is referred as Technical Debt (TD), and recently it has been recognized as a useful basis for the development of theoretical and practical frameworks [2]. Tom et al. [3] have explored the TD metaphor and outlined a first framework in 2013. Part of the overall TD is to be related to architecture sub-optimal decisions, and it is regarded as Architecture Technical Debt (ADT) [4]. More precisely, ATD is regarded as implemented solutions that are sub-optimal with respect to the quality attributes (internal or external) defined in the desired architecture intended to meet the companies' business goals.

ATD has been recognized as part of TD in a recent (2015) systematic mapping study on TD [4]. However, such study highlights several deficiencies in the current body of knowledge: lack of reliable industrial studies, lack of focus on architecture anti-patterns and lack of studies involving the whole TD management process. In this paper we aim at filling such current gaps by investigating, in several companies, the overall phenomenon of accumulation and refactoring of ATD. The study of such subject would also contribute to ASD frameworks, by highlighting activities for enhancing agility in the task of developing and maintaining software architecture in large projects [5].

* Corresponding author.
E-mail addresses: antonio.martini@chalmers.se (A. Martini), jan.bosch@chalmers.se (J. Bosch), michel.chaudron@cse.gu.se (M. Chaudron).

http://dx.doi.org/10.1016/j.infsof.2015.07.005
0950-5849/

In the context of large-scale ASD, our research questions are:

RQ1: What factors cause the accumulation of ATD?
RQ2: How is ATD accumulated and refactored over time?
RQ3: What possible refactoring strategies can be employed for managing ATD?

In this paper we have employed a 18 months long, multiple-case study involving 7 different sites in 5 large Scandinavian companies in order to shed light on the phenomenon of accumulation and refactoring of ATD. We have analyzed the qualitative data obtained from more than 50 h of formal interviews complemented with continuous informal meetings with key roles involved in the architectural work, using a combination of inductive and deductive approach proper of Grounded Theory. We have qualitatively developed and evaluated a taxonomy of the factors to inform RQ1 and a set of models to inform RQ2. We have also derived some preliminary conclusions on which refactoring strategies can be applied and what effects they have to inform RQ3.

The main contributions of the papers are:

- A taxonomy of the causes for ATD: we present the factors for the explanation of the phenomena such as accumulation and refactoring of ATD. These factors might be studied and treated separately, and offer a better understanding of the overall phenomenon.
- Two qualitative models of the trends in accumulation and refactoring of ATD over time.
  - Crisis model – Shows the strictly increasing trend of ATD accumulation and how it eventually reaches a crisis point. We describe the evidences related to the occurrence of the crisis point and we connect such phenomenon to the different factors and their influence on the accumulation.
  - Phases model – Shows when ATD is currently accumulated and refactored during different software development phases. It helps identifying problem areas and points in time for the development of practices that would 1) avoid accumulation of ATD and/or 2) ease the refactoring of ATD. Such practices would be aimed at delaying the crisis point.
- Possible refactoring strategies: we analyze how different refactoring strategies might lead to best- and worst-case scenarios with respect to crisis points.
- A detailed description of an additional and in-depth industrial case, which contributes to empirically evaluate the factors and to analyze the relationships among them in a specific context.

The rest of the paper is structured as follows: Section 2 gives the reader more references and background on ATD and on the conceptual framework used in this study. In Section 3 we explain our research design: overall design, description of the cases, methods for data collection and analysis and evaluation of results. In Section 4 we list the results: factors causing ATD, models of accumulation of ATD over time, possible refactoring strategies, description of the in-depth case-study and the evaluation of results. In Section 5 we examine how the results inform the RQs, we discuss practical and theoretical implications of this study and we discuss the degree of validity of each result. We also point at limitations and open issues for future research, and we discuss the related work. We summarize the conclusions in Section 6.

2. Architecture and Technical Debt

2.1. Definition of ATD

ATD is regarded [3] as "sub-optimal solutions" with respect to an optimal architecture for supporting the business goals of the organization. Specifically, we refer to the architecture identified by the software and system architects as the optimal trade-off when considering the concerns collected from the different stakeholders. In the rest of the paper, we call the sub-optimal solutions *inconsistencies* between the implementation and the architecture, or *violations*, when the optimal architecture is precisely expressed by rules (for example for dependencies among specific components). However, it is important to notice that (in our studied cases) such optimal trade-off might change over time, as explained in this paper, due to business evolution and to information collected from implementation details. Therefore, it is not correct to assume that the sub-optimal solutions can be completely identified and managed from the beginning.

In the next section, we mention some classes of examples of what can be considered ATD. Such examples are extracted from another paper by the same authors on the same subject [7].

2.2. Examples of ATD

2.2.1. Dependency violations

The presence of architectural dependencies (for example at different component levels) which are considered forbidden in the (context-specific) architecture can be considered ATD. An example of this class of items is represented by a component that, when executed, should not trigger the execution of another component, as specified by the architects/architecture. A study on this problem has been conducted also by Nord et al. [8].

2.2.2. Non-uniformity of patterns and policies

Patterns and policies defined by the architecture (and the architects) might not be kept consistent through the system. For example, there might be a name convention applied in part of the system that is not followed in another part of the system. Another example is the presence of different design or architectural patterns used to implement the same functionality, such as different interaction patterns used among different components (we intend those differences that are not motivated by precise design constraints).

2.2.3. Code duplication (non-reuse)

Quite recognized in literature and in practice is the presence of very similar code (if not identical) in different parts of the system, especially in different products, not grouped into a reused component.

2.2.4. Temporal properties of inter-dependent resources

Some resources might need to be accessed by different part of the system (for simplicity we will take the example of having different components accessing the same resource). In these cases, the way in which a component interacts with the resource might change the interaction of other components with the same resource. This aspect is especially related to the temporal dimension: for example, the order with which two components change the status of a database or access it, would change the behavior of the system. For this reason, specific scheduling patterns can be designed, which are used for assuring the correct interaction of components. As a concrete example, we can mention the convention of having only synchronous calls to a certain component. An ATD item of this category is represented by the non-application of such patterns by some developers or teams.

2.2.5. Sub-optimal mechanism for non-functional requirements

Some non-functional requirements, such as scalability, performance or signal reliability, need to be recognized before or early during the development and need to be tested. The ATD items represent the lack of an implementation that would assure the

satisfaction of such requirements, but also the lack of mechanisms for them to be tested.

2.3. Previous research on ATD

The term *Technical Debt* (TD) has been first coined at OOPSLA by Cunningham [9] to describe a situation in which developers take decisions that bring short-term benefits but cause long-term detriment of the software. The term has recently been further studied and elaborated in research: in 2013 Tom et al. [3] conducted an exploratory case study technique that involves multi-vocal literature review, supplemented by interviews, in order to draw a first categorization of TD and the principal causes and effects. In such paper we can find the first mentioning of Architectural Technical Debt (ATD, categorized together with Design Debt). A further classification can be found in Kruchten et al. [2], where ATD is regarded as the most challenging TD to be uncovered since there is a lack of research and tool support in practice. Finally, ATD has been further recognized in a recent systematic mapping [4] on TD. Such recent research highlights the gap in the current scientific knowledge, which gives us the motivation for this work.

2.4. Previous research on management of TD

Some studies have been conducted on the management of TD, also supported by a dedicated workshop (MTD), usually co-located with premium conferences, such as ICSE and ICSME.

A first roadmap has been created in 2010 by Brown et al. [10]. In 2011 Guo et al. proposed an initial portfolio approach with the creation of TD *items*. The same authors proposed a further empirical study on tracking TD [11], while Seaman et al. identified the theoretical importance of TD as risk assessment tool in decision making [12]. TD has also been used for defining part of a method for assessing software quality, SQALE [13]. Such model has been also implemented in a tool, but the main support is currently given on a source code level (very limited on the ATD aspect).

2.5. Models for technical debt

The studies in TD are quite recent, and the subject is not mature. Some models, empirical [14] or theoretical [15] have been proposed in order to map the metaphor to concrete entities in software development. We use, in this paper, a conceptual model comprehending the main components of TD, as described in the next sections.

2.5.1. Debt

The debt is regarded as the actual technical issue. Related to the ATD in particular, we consider the ATD item as a specific instance of the implementation that is sub-optimal with respect to the intended architecture to fulfill the business goals. For example, referring to Section 2.2, a possible ATD item is a dependency between components that is not allowed by the architectural description or principles defined by the architects. Such dependency might be considered sub-optimal with respect to the *modularity* quality attribute [16], which in turn might be important for the business when a component needs to be replaced in order to allow the development of new features.

2.5.2. Principal

The principal is the cost for refactoring the specific TD item. In the example case explained before, in which an architectural dependency violation is present in the implementation, the principal is the cost for reworking the source code in order to have the dependency removed and the components not being dependent from each other.

2.5.3. Interest

A sub-optimal architectural solution (ATD) might cause several effects (for example, as described in [7]), which have an impact on the system, on the development process or even on the customer. For example, having a large number of dependencies between a large amount of components might lead to a big testing effort (which might represent only a part of the whole interest in this case) due to the spread of changes. Such effect might be paid when the features delivered are delayed because of the extra time involved during continuous integration. In this paper, we treat accumulation and refactoring of ATD as including both the principal and the interest.

2.6. The time perspective

The concept of TD is strongly related to time. Contrarily to having absolute quality models, the TD theoretical framework instantiates a relationship between the cost and the impact of a single sub-optimal solution. In particular, the metaphor stresses the short-term gain given by a sub-optimal solution against the long-term one considered optimal. Time wise, the TD metaphor is considered useful for estimating if a technical solution is actually sub-optimal or might be optimal from the business point of view. Such risk management practice is also very important in the everyday work of software architects, as mentioned in Kructhen [17] and Martini et al. [18]. Although research has been done on how to take decisions on architecture development (such as ATAM and ALMA [19]), there is no empirical research about how sub-optimal architectural solutions (ATD) are accumulated over time and how they can be continuously managed.

3. Research design

We preformed a 18-monhts long, multiple-case, embedded case study involving 7 Scandinavian sites in 5 large international software development companies. We decided to collect data from many large companies (multiple-case), in order to increase the degree of source triangulation [20]. The reasons for conducting an embedded study (including more than one case within the same context, see company C in the "Case description" section) was to maximize the number of confirmatory evidences and to strengthen the internal validity of the results with respect to the chosen company.

The nature of the study was exploratory, since the lack of previous literature focusing on the specific research problem. Therefore, we wanted to maximize the coverage of possible software companies within the boundaries defined by the key characteristics of *large* and *Agile*, in order to capture as many ATD items experienced by the companies, but at the same time having the opportunity to dig out as many details from the context as possible. The research design is outlined in Fig. 1. In such picture, we show phases of data collection (black boxes), data analysis (white rectangles), results (grey ellipses) and the variable used for axial coding (*Relationship over time*). The arrows, where not explicitly specified, represent the flow of information from an activity to a specific result or vice versa.

We conducted a first explorative study (phase I), followed by retrospective sessions (phase II), a set of cross-company evaluative interviews (phase III) and finally a follow-up case-specific study (phase IV). The retrospective sessions started with the investigation of the most effortful events faced by the companies in the recent past and tracking them to the source of the problem, which in some cases would lead to ATD items accumulated in the system. Since retrospective studies usually provide, as a result of the data collection, more emphasis on the challenges currently faced by the interviewees, we included organizations being in different phases of the development, which would experience different occurrences of ATD instances. For example, one organization was

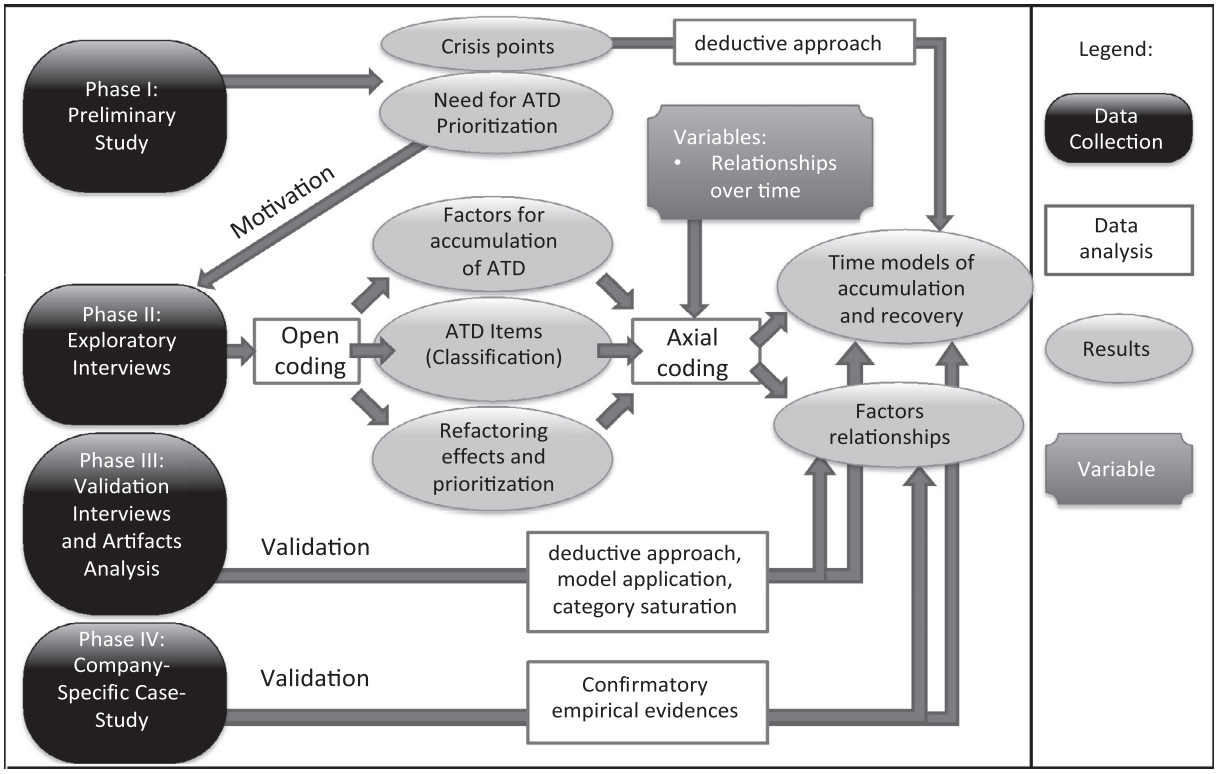

Fig. 1. Our research design.

currently experiencing a crisis, while others where in the middle of big refactorings and others where mainly focused in feature development.

3.1. Case description

3.1.1. Companies description

Company A carried out part of the development out by suppliers, some by in-house teams following Scrum. The surrounding organization follows a stage-gate release model for product development. Business is driven by products for mass customization. The specific unit studied provides a software platform for different products. The internal releases were short but needed to be aligned, for integration purposes, with the stage gate release model (several months).

In company B, teams work in parallel in projects: some of the projects are more hardware oriented while others are related to the implementation of features developed on top of a specific Linux distribution. The software involves in house development with the integration of a substantial amount of open source components. Despite the Agile set up of the organization, the iterations are quite long (several months), but the company is in transition towards reducing the release time.

Customers of Company C receive a platform and pay to unlock new features. The organization is split in different units and then in cross-functional teams, most of which with feature development roles and some with focus on the platform by different products. Most of the teams use their preferred variant of ASD (often Scrum). Features were developed on top of a reference architecture, and the main process consisted of a pre-study followed by few (ca. 3) sprint iterations. The embedded cases studied slightly differed: C3 involved globally distributed teams, while the other units (C1 and C2) teams were mostly co-located.

Company D is a manufacturer of a product line of embedded devices. The organization is divided in teams working in parallel and using SCRUM. The organization has also adopted principles of software product line engineering, such as the employment of a reference architecture. Also in this case, the hardware cycle has an influence.

Company E is a company developing software for calculating optimized solutions. The software is not deployed in embedded systems. The company has employed SCRUM with teams working in parallel. The product is structured in a platform entirely developed by E and a layer of customizable assets for the customers to configure. E supports also a set of APIs for allowing development on top of their software.

3.1.2. Commonalities and differences in the studied companies

All the companies have adopted a component based software architecture, where some components or even entire platforms are re-used in different products. The language that is mainly used is C and C++, with some parts of the system developed in Java and Python. Some companies use a Domain Specific Language (DSL) to generate part of their source code.

All the companies have employed SCRUM, and have a (internal) release cycle based on the one recommended in SCRUM. However, the embedded companies (A–D) depend on the hardware release cycles, which influence the time for the final integration before the releases. Therefore, some of the teams have internal, short releases and external releases according to the overall product development.

3.2. Data collection

We have employed a 4-phase investigation of the ATD items and effects. The four phases (black boxes) and their results (the outcome of the phases, highlighted with arrows connected to elliptic boxes) are visible in Fig. 1. Table 1 shows several properties of our data collection: for each phase (explained in details below),

the total number of participants, the total amount of hours recorded from the interviews, the number of sessions, if the sessions were company-specific or cross-company and which roles were part of the investigation.

We have conducted a case-study following the guidelines in [20]. We relied especially on semi-structured interviews supported, when possible, by existing architecture documentation in order to provide multiple sources of evidence. Interviews with the architects assured the best representation of the desired architecture for the given case, since the existing documentation was usually not updated or not sufficient for analyzing if ATD was in place. Causes and effects related to the ATD also needed to be reported by the employees directly involved in the case studied. We have asked about possible archival data to analyze, for example source code commits and project data, but we have not found reliable measures to track the extra-effort (interest) related to the studied ATD. Nevertheless, we have surveyed many different roles involved in the studied cases, in order to be able to compare different perspectives on the same case and to mitigate the bias related to only one reporting person.

We have not followed the Grounded Theory strategy for data saturation [21], for two main reasons: we did not aim at a complete saturation of the categories, since we assumed that we might have encountered different situations in different settings and we might have never reached such status. The second reason was because we could not afford to reach complete saturation from a resources point of view, since the interviews conceded by the companies were too limited to be used for complete saturation. Nevertheless, we have used the principle discussed by Yin in [22], in which the balance between resources and findings is set when the researchers obtain confirmatory evidence of the findings. We report, in the discussion session, a table in which we show which results can be considered confirmed and which ones need further investigation.

Phase I – We started with a preliminary study involving 3 of the abovementioned cases, in particular A, C_1, and C_2, in which we explored the needs and challenges of developing and maintaining architecture in an Agile environment in the current companies. We organized three multiple-participant interviews at the different sites involving several roles. The combined interviews lasted 4 h and involved developers, testers, architects responsible for different levels of architecture (from low level patterns to high level components) and product managers. The results from the first iteration were evaluated and discussed in a final one-day workshop involving 40 representatives from all the 7 cases (see Table 1).

The preliminary study showed a major challenge in managing Architectural Technical Debt (ATD) and its economical implications related to costs and time. In particular, the studied companies emphasized the struggle, rather than in identifying the debt, in estimating its impact and therefore in prioritizing the items among themselves and comparing the ATD items against features (*Need for ATD prioritization* in Fig. 1). During such investigation, we also collected several instances of what was recognized as "*crisis points*": the companies described such crisis as occurring almost cyclically, points in time when development was particularly slowed down or even stopped until a refactoring took place. From the data we developed the crisis point model explained in Section 4.2.1.

Phase II – In the second phase we conducted 7 sets of interviews, one set for each company (Table 1). Each set lasted a minimum of 2 h, and we included participants with different responsibilities, in order to cover many aspects: the source of ATD (developers), the architectural implications (architects and system engineers), the prioritization decisions taken (product owners) and also the stakeholders of the effects (we included also testers and developers involved in maintenance projects when assigned to a dedicated project).

The formal interviews were also complemented with the preliminary study of software architecture documentation for each case, to which we could map the mentioned ATD items. The collaboration format allowed the researchers to conduct ad hoc consultations, several hours of individual and informal meetings with the chief architects (at least one per company) responsible for the documentation and the prioritization of ATD items. Such activity was conducted especially for further explanations, follow-up questions and evaluation of the developed models. The main rationale for meeting with architects was that they are the main stakeholders involved in the prioritization of ATD, both for their prioritization with respect to the teams' backlogs, but also for their prioritization "against" the features with the product owners.

Each set of interviews followed a process designed to identify important architecture inconsistencies (ATD) that needed to be tracked because of their impact in decreasing developing time. We started with a plenary session where we briefly introduced

Table 1
Data collection: in the table is possible to see the various phases, each of which included a number of participants covering different roles, either from one specific company or from several companies. The table also shows how many sessions have been conducted for each phase, and how many total hours the sessions lasted.

Phases of data collection	Number of participants	Total of hours recorded	Number of sessions	Companies involved in each session	Roles involved
Phase I (Preliminary interviews)	25	12	3	Company-specific	Developers, architects, testers, line managers, Scrum m.
Phase I (Evaluation workshop)	40	4	1	Cross-company	Developers, architects, line managers
Phase II (group interviews)	26	14	7	Company-specific	Developers, architects, product owners
Phase II (Evaluation workshop)	10	3	2	Cross-company	Architects, line managers
Phase III (Evaluation interviews 1)	10	4	1	Cross-company	Architects, product owners
Phase III (Evaluation interviews 2)	12	4	1	Cross-company	Architects, developers, scrum masters
Phase III (Evaluation workshop)	20	4	2	Cross-company	Architects
Phase IV (Detailed case-study)	7	6	3	Company-specific	System architect, software architect, scrum master, developers
Informal interaction	7	NA	NA	Company-specific	Software and system architects

what ATD is, using references from several sources included in this paper (e.g. the purpose of tracking ATD, [12]), which were also provided to the informants in advance. We took a retrospective approach, in order to identify real cases happened in the recent past rather than rely on speculations about what could happen in the future:

- we asked about major refactorings and high effort perceived during feature development or maintenance work leading to architecture inconsistencies
- we investigated the causes for the identified inconsistencies (ATD).
- we asked to explain the current process of identification of architecture inconsistency.
- we asked how the ATD refactoring was prioritized. In particular, we asked when it was prioritized as important or when it was postponed and not included in the next development plan (in the following, we will refer to these two cases as *high-prioritized* or *low-prioritized* refactoring).

The strength of this technique relies on finding the relevant (more costly) architecture inconsistencies (ATD) by identifying their worst effects first, instead of listing a pool of all the possible inconsistencies and then selecting the relevant ones. We have found no other studies applying such "reverse" technique, which add methodological novelty to the current results.

Phase III – The third phase consisted of two evaluation activities: we organized 3 multiple-company group interviews, including all the roles involved in the investigation, developers, architects and product owners, where we showed the models for their recognition and improvement. For example, we proposed the crisis model and we asked, when recognized, to strengthen the model with further concrete and real examples. For example, in one case we could see (but we cannot report the picture for confidentiality reasons) the burn-down chart for feature development "abnormally interrupted" due to the crisis point. We also included, where possible, the analysis of artifacts such as lists of *Technical Issues* or *Architectural Improvement* identified within the company. Such deductive procedure strengthened the inductive process employed in the first and second phases, providing several confirmatory evidences.

As a further evaluation step we organized 2 plenary workshops with around 20 participants also from 2 other large companies not previously participating in the study, in order to further strengthen the external validity of the results.

Phase IV – Finally, we followed-up the workshop with a company-specific case study. By studying a concrete example in-depth, we aimed at mapping the factors to a typical example and understanding the relationships among the different factors for the specific context. Such technique is recognized as one of the most effective in qualitative research [20,22] and is called *pattern matching*. In order to study such case, we set up a 2-h interview with the system architect, the software architect and the scrum master of the involved team. We then followed-up with some team members, 4 developers together with the scrum master. We specifically studied the context of the case described here as C_2. Rather than showing the model to the participants in advance, we preferred to start the investigation from the narrative of a recent ATD case causing effort. We then proceeded with asking specific questions in order for the researchers to recognize the factors and to understand their relationships, in order to match the previously identified patterns. The difference between this case and the ones collected during Phase II consists of the level of details. In this case, we did not aim at having a collection of cases but rather at gaining as many details as possible in order to provide a better understanding of the specific case.

3.3. Data analysis

The workshops were recorded and transcribed. The analysis was conducted following an approach based on Grounded Theory [21].

Given the exploratory nature of our study and the need to develop novel concepts and theories about ATD, we opted for using the following methods, frequently employed for the analysis of large amount of semi-structured, qualitative data representing complex combinations of technical and social factors. The analysis followed the steps highlighted in Fig. 1.

Open Coding – In phases I and II (the exploratory ones) the first step was to analyze the data in search for emergent concepts following open coding, which would bring novel insights on the analyzed issue. For example, we did not know in advance that *non-completed refactoring* would lead to new ATD. Without the open coding but just looking for predefined categories would have probably caused us to miss this novel concept. Then we categorized the codes using the taxonomies developed during the pre-study. For example, we used *causes* as a pre-defined category. Codes within the same category were also grouped in order to create concepts: for example, within the *causes* categories we have found 34 codes, which were grouped in higher level codes (or concepts), which are represented in the factors outlined in Section 4.1.

We used a Qualitative Data Analysis (QDA) tool for open coding, Atlas.ti. Such tool gives support to keep track of the links between categories, codes and the quotations they are grounded to (providing, as recommended in [20] a *chain of evidence*). A screenshot with explanations is reported in Fig. 2, showing the elements used in the analysis (*quotations*, *codes* and *categories*) and how they are connected through the tool. The initial codes were 144, of which 97 were selected as relevant for the RQs in this paper. We then merged the redundant codes and filtered them by their *groundedness* (based on how many quotation they were based on), selecting only the ones that had at least 2 quotations (i.e., having *confirmatory evidences*): the final number of codes was 38. The predefined categories were decided by the three authors altogether. The open coding was performed by the first author, and the codes were checked by the other authors periodically.

Axial Coding – The codes and categories were compared through axial coding in order to highlight connections orthogonal to the previous developed categories. Since ATD is strictly connected with time, we have used the *time variable* as axis for axial coding. The aim was to understand relationships (e.g. causality, dependencies, etc.) among the factors. This step showed the presence of sequences or patterns of accumulation factors for ATD over time. In particular, such analysis produced the model for accumulation and refactoring of ATD explained in Section 4.2, and the same analysis on the in-depth case study was done to conduct *pattern matching* with evaluation purposes (see Section 4.5.2). The axial coding was carried out by the first author supervised by the second author, while the third author checked the results.

For this research activity, we needed a tool for visualizing the factors and the refactoring of ATD over time. We therefore exported the related codes and concepts from Atlas.ti into Microsoft PowerPoint, which allows the easy creation of a graphical timeline. In order to place the factors on the timeline, we used the time information contained in the quotations linked to the factors: for example, the informants mentioned that the *uncertainty about the use cases* was present *in the beginning of the development* and that the *urgency* was present especially *when close to a release*. Some of the factors, for which there was no specific time but they were mentioned as being always on going, we represented them as spanning from the development start to the release. As for those factors that were not bound to time but were happening in an ad-hoc fashion (for example, *non-completed refactoring*) were

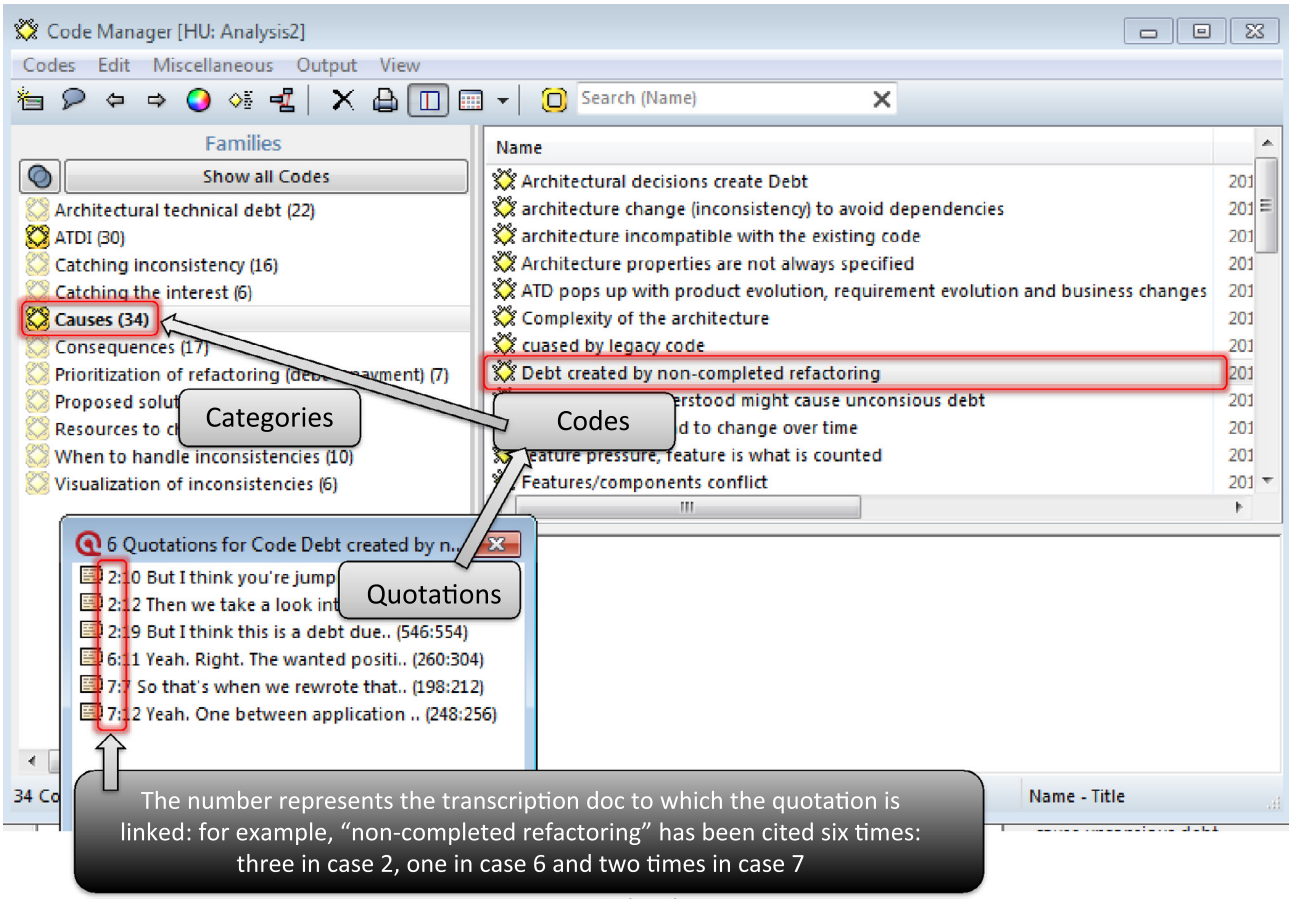

Fig. 2. Screenshot of the QDA tool and the chain of evidences.

omitted in the model since they were not part of a recurrent pattern. The same approach was used for the concepts related to the prioritization of ATD refactoring: for example, the quotations from the informants usually reported the refactoring as postponed with respect to the release. Therefore we represented the refactoring activities as the decrement of ATD happening after the release point. The outcome of axial coding, the model of accumulation and refactoring over time, is visible in Fig. 4.

Deductive Approach – We performed deductive analysis especially for evaluation purposes, but also in order to relate two different models. For example, after the open coding analysis phase, the factors and the models were deductively checked against the overall model of crisis point, developed and evaluated during the first phase of the research in order to understand if detailed models could fit and explain the overall one. This last analysis step showed the results in Section 4.2.2).

In particular, we projected the model showed in Fig. 4 over a longer time span, taking three cases with respect to three different refactoring strategies: when all the ATD is removed, when ATD is partially removed and when is not removed at all (Figs. 5, 6, and 7). Then we have related these three options with respect to the crisis point model in Fig. 3. The outcome of the projection combined with the crisis point model led to the hypothesis described in Fig. 8, where different refactoring strategies would lead to different crisis points.

The deductive approach was also used during interviews in the evaluation phase III, when we used the models for eliciting concrete cases to confirm (or reject) the inductively obtained hypothesis made during phases I and II. An example can be found in Section 4.5.2.1, where we report concrete quotes answering the evaluation questions on the Crisis Model.

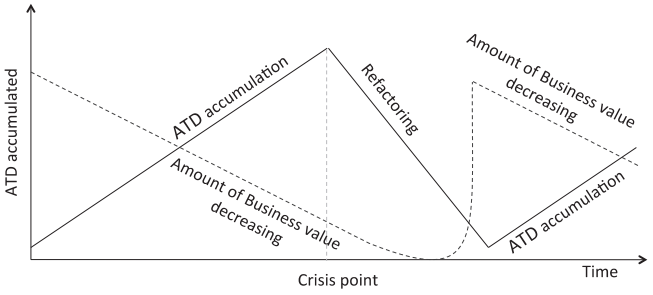

Fig. 3. Crisis point model.

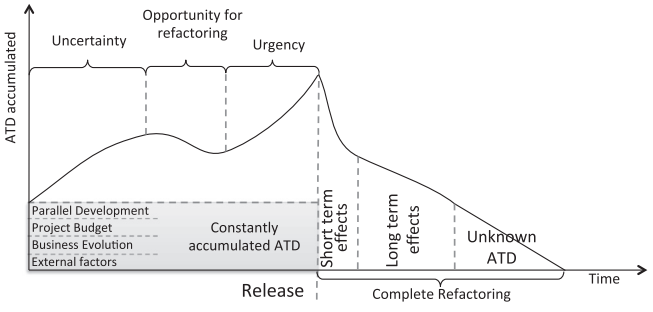

Fig. 4. Factors of constantly accumulated ATD, phases and kinds of refactorings.

In phase IV, the deductive approach was used during data collection, when we were asking questions in order to recognize, in the in-depth case study, the factors found in phase I and II. More

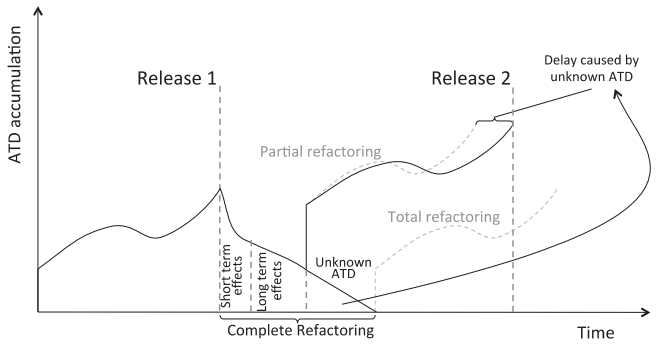

Fig. 5. Refactoring maximization: some ATD is always accumulated and impacts next development (e.g. release 2). *Total Refactoring* is not realistic in practice. The best option is *Partial Refactoring*, when short-term and long-term ATD is removed.

specifically, we have used the *pattern matching* analytical strategy recommended in [22]. Such strategy is used to verify the existence of previously formulated patterns (hypotheses) after a new collection of data. In our cases, we used the models obtained by the data collection conducted during phase I and II as *pattern tests* to be used during phase IV to reveal the pattern. This kind of evidence contributed in evaluating the formulated models (with different strength for different results, as explained in Section 4.5.2). In this case we used a similar practical approach used for Axial Coding, placing the factors in a timeline. This way, we had two similar (with the same factors over time) models to compare, one from the inductive analysis of phases I and II and one from the case study: the comparison would lead to the matching (or not) of the found patterns.

3.4. Factors and models evaluation

As explained in Fig. 1 we conducted two steps of evaluation (phase III and IV). The crisis point model (outlined in Section 4.2.1) was developed during phase I. It was qualitatively evaluated during phase II, where *all* the informants recognized the model as representing the facts in their company (when we actually managed to reach data saturation for the crisis point model, as recommended in the Grounded Theory approach). During phase III and IV we also probed the crisis model by asking more precise questions on the evidence of such a crisis model. In one case, the participants showed us a burn-down chart for feature development "abnormally interrupted" (citing from the whiteboard) due to the crisis. Unfortunately, we cannot report the actual source for confidentiality reasons.

The models for accumulation and refactoring of ATD were developed in phase II and took as input the deductive model about the crisis point developed previously. The new models were then evaluated during the evaluation workshop of phase II and during phase III.

The factors were evaluated both during phase III and by *pattern matching* during to the in-depth case studied in phase IV. Such technique is recognized as one of the most effective in qualitative research [20,22].

4. Models of accumulation and refactoring of Architecture Technical Debt

We have divided the results in four parts: first we highlight the causes for ATD accumulation (factors). Then we use such factors to describe a model for accumulation and refactoring of ATD over time. We then narrate in a chronological sequence of events an additional industrial case, which shows the factors and their relationships over time. We finally show results from the evaluation process of the factors and models, both with quotations and matched patterns.

4.1. Causes of ATD accumulation (factors)

4.1.1. Business factors

4.1.1.1. Business evolution creates ATD. The amount of customizations and new features offered by the products brings new requirements to be satisfied. Whenever a decision is taken to develop a new feature or to create an offer for a new customer, instantaneously the desired architecture changes and the ATD is automatically created. The number of configurations that the studied companies need to offer simultaneously seems to be growing steadily. If for each augmentation of the product some ATD is automatically accumulated when the decision is taken, the same trend of having more configurations over time implies that the corresponding ATD is also automatically accumulated faster.

4.1.1.2. Uncertainty of use cases in early stages. The previous section also suggests the difficulty in defining a design and architecture that has to take in consideration a lot of unknown upcoming variability. Consequently, the accumulation of inconsistencies towards a "fuzzy" desired design/architecture is more likely to take place in the beginning of the development (for example, during the first sprints).

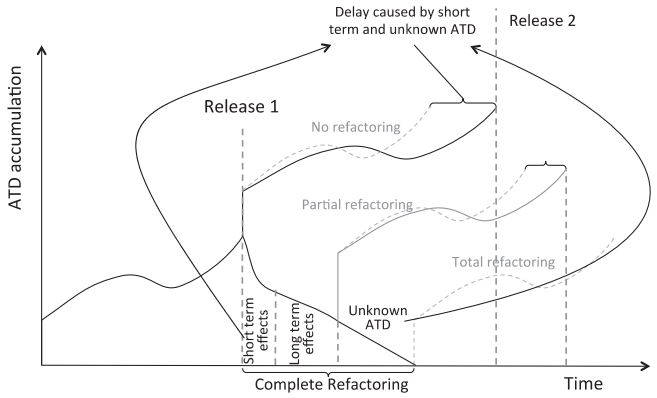

Fig. 6. Refactoring minimization: when *No refactoring* is applied, both short-term and unknown ATD would impact release 2. However, there might be a short-term benefit with respect to doing long-term refactoring (difference with *Partial refactoring*).

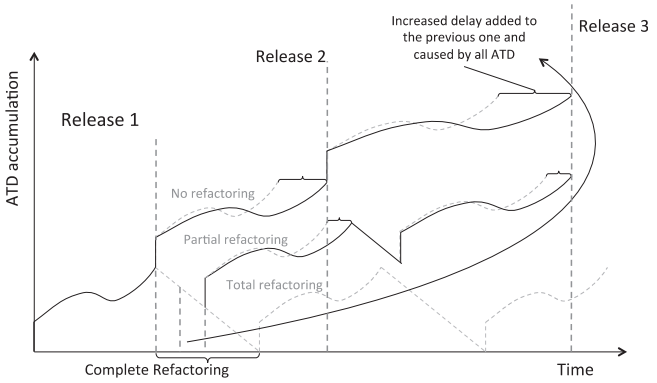

Fig. 7. Comparison of refactoring strategies: after a number of features released (symbolically 3 in the picture), the long-term ATD starts to have an impact, decreasing the advantages of the short-term benefits given by the *No refactoring* strategy.

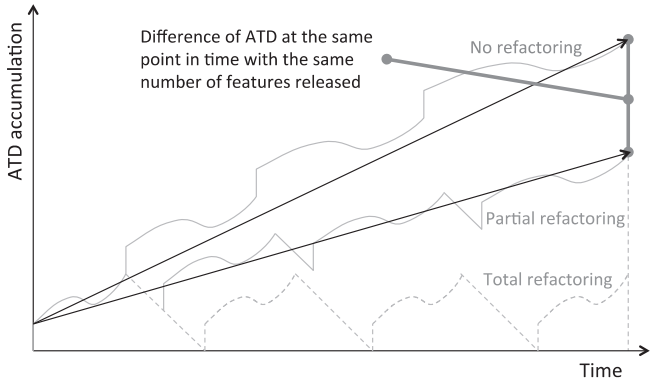

Fig. 8. Comparison of refactoring strategies: after a number of features released (symbolically 3 in the picture), the time elapsed to deliver would be the same but the ATD present in the system would be more in the *No recovery* strategy, which will affect long-term development.

4.1.1.3. Time pressure: deadlines with penalties. Constraints in the contracts with the customers such as heavy penalties for delayed deliveries make the attention to manage ATD less of a priority. The approaching of a deadline with a high penalty causes both the accumulation of inconsistencies due to shortcuts and the low-prioritization of the necessary refactoring for keeping ATD low. The urgency given by the deadline increases with its approaching, which also increases the amount of inconsistencies accumulated.

4.1.1.4. Priority of features over product architecture. The prioritization that takes place before the start of the feature development tends to be mainly feature oriented. Small refactorings necessary for the feature are carried out within the feature development by the team, but long-term refactorings, which are needed to develop "architectural features" for future development, are not considered necessary for the release. Moreover, broad refactorings are not likely to be completed in the time a feature is developed (e.g. few weeks). Consequently, the part of ATD that is not directly related to the development of the feature at hand is more likely to be postponed.

4.1.1.5. Split of budget in project budget and maintenance budget boosts the accumulation of debt. According to the informants, the responsibility associated only with the project budget during the development creates a psychological effect: the teams tend to accumulate ATD and to push it to the responsible for the maintenance after release, which rely on a different budget.

4.1.2. Design and architecture documentation: lack of specification/ emphasis on critical architectural requirements

Some of the architectural requirements are not explicitly mentioned in the documentation. This causes the misinterpretation by the developers implementing code that is implicitly supposed to match such requirement. According to the informants, this is also threatening the refactoring activity and its estimation: the refactoring of a portion of code for which requirements were not written (but the code was "just working", implicitly satisfying them) might cause the lack of such requirements satisfaction.

As an example, three cases have mentioned temporal-related properties of shared resources. A concrete instance of such a problem is a database, and the design constraint of making only synchronous calls to it from different modules. If such requirement is not specified, it may happen that the developers would ignore such a constraint. In one example made by the informants, the constraint was violated in order to meet a performance requirement important for the customer, but subsequently such choice created several bugs that were difficult to catch (because of the temporal nature of the pattern), which caused a number of delays during maintenance. This is also connected with Section 4.1.1.4.

4.1.3. Reuse of legacy/third party/open source

Software that was not included when the initial desired architecture was developed contains ATD that needs to be fixed and/or dealt with. Examples included open source systems, third party software and software previously developed and reused. In the former two cases, the inconsistencies between the in-house developed architecture and the external one(s) might pop up after the evolution of the external software.

4.1.4. Parallel development

Development teams working in parallel automatically accumulate some differences in their design and architecture. The Agile-related empowerment of the teams in terms of design seems to amplify this phenomenon. An example of such phenomenon mentioned as causing efforts by the informants are the naming policy. A name policy is not always explicitly and formally expressed, which allows the teams to diverge or interpret the constraint. Another example is the presence of different patterns for the same solution, e.g. for the communication between two different components. When a team needs to work on something developed by another team, this non-uniformity causes extra time.

4.1.5. Uncertainty of impact

ATD is not necessary something limited to a well-defined area of the software. Changing part of the software in order to improve some design or architecture issues might cause ripple effects on other parts of the software depending on the changed code. Isolating ATD items to be refactored is difficult, and especially calculating all the possible effects is a challenge. Part of the problem is the lack of awareness about the dependencies that connect some ATD to other parts of the software. Consequently, there exists some ATD that remains *unknown*.

4.1.6. Non-completed refactoring

When refactoring is decided, it is aimed at eliminating ATD. However, if the refactoring goal is not completed, this not only will leave part of the ATD, but it will actually create new ATD. The concept might be counter-intuitive, so we will explain with an example. A possible refactoring objective might be to have a new API for a component. However, what might happen is that the new API is added but the previous one cannot be removed, for example because of unforeseen backward compatibility with another version of the product. This factor is related to other two: time pressure might be the actual cause for this phenomenon, when the planned refactoring needs to be rushed due to deadlines with penalties (see Section 4.1.1.3) and the effects uncertainty (see Section 5), which causes a planned refactoring to take more time than estimated because of effects that have been overlooked when the refactoring was prioritized.

4.1.7. Technology evolution

The technology employed for the software system might become obsolete over time, both for pure software (e.g. new versions of the programming language) and for hardware that needs to be replaced together with the specific software that needs to run on it. The (re-)use of legacy components, third party software and open source systems might require the employment of a new or old technology that is not optimal with the rest of the system.

4.1.8. Lack of knowledge

Software engineering is also an individual activity and the causes for ATD accumulation can also be related to sub-optimal decision taken by individual employees due to:

4.1.8.1. Inexperience. New employees are more subjected to accumulating ATD due to the natural non-complete understanding of the architecture and patterns.

4.1.8.2. Lack of domain knowledge. This factor might be related to the previous one, or, as the informants mentioned, to the generalization of Agile teams, which might need to develop a feature accessing a complex component of which they do not have expertize.

4.1.8.3. Ignorance. Lack of knowledge about where the architectural rules are stored (documentation).

4.1.8.4. Carelessness. Lack of awareness of the importance of architecture. A recurrent statement from the informant is that having documentation is not enough to avoid architecture violations.

4.2. ATD accumulation and refactoring models

Using the previously listed factors for ATD accumulation and the data on refactoring prioritization, we modeled the evolution of ATD over time with respect to the overall speed of adding features and to one specific release. The values in the pictures are only aimed at visualizing the trends perceived by the informants, and they do not represent any exact values. We have chosen the "function" format since it would explain the results in a more visual way.

4.2.1. Crisis-based ATD management

The current management of ATD is driven by a crisis (Fig. 3). The informants explain that the ATD usually grows (black continuous line in the picture) until the effect makes adding new business value so slow (dashed line in the picture) that it becomes necessary to conduct a big refactoring or even rebuilding a platform from scratch. The usual approach is to wait for such event with limited monitoring and limited reduction of ATD growth during development. In fact, the long-term improvement is considered risky invested time.

4.2.2. ATD accumulation and refactoring trends during feature development

In Fig. 4 we present the various phases of ATD accumulation over time, on the left part of the graph, and the hypothetical refactoring ("complete refactoring") of ATD on the right, divided by different kinds of identified ATD.

4.2.2.1. Constant ATD accumulation. From the analysis of the factors, we understood that part of the ATD is constantly accumulated over a release (grey area in Fig. 4). Such part is composed by several components described previously in Section 4.1: *business evolution*, *parallel development* and *project budget* are the ones that are most connected with the companies' direct decisions, whilst other factors are external to the company (for example, the change of an Open Source module developed by third party or the *technology evolution*). Some of them might be considered as multipliers for the other kinds of ATD, but for simplicity and for lack of more precise measures, we treat them as constants in the graph.

4.2.2.2. Phases of ATD accumulation. According to the informants, when the feature development starts, there is a certain degree of *uncertainty* that tends to decrease over time. Since ATD is created when there is uncertainty (see Sections 4.1.1.1 and 4.1.5), the curve on the graph in Fig. 4 representing ATD accumulation tends to raise in the beginning until the team has a clearer understanding of the requirements, desired design and desired architecture altogether. At this point, the hypothesis is that ATD accumulation would slow down. The ATD starts again being accumulated abundantly when the *urgency* for meeting the deadline shows up in the team. Urgency seems to grow constantly with the deadline approaching, causing the level of ATD to grow accordingly. We do not know exactly from the data when uncertainty stops and urgency starts and if the two phases overlap. Some informants mention a time window when the team refactors part of the ATD needed to deliver the feature, but it is unlikely that all the accumulated ATD is refactored during this phase (especially the constant one). However, this seems to be a good *opportunity for refactoring* in the process when the team might decide to take care of the ATD before the release. The *project budget* factor (Section 4.1.1.5) might have a negative impact on such practice though, demotivating the team to look for such opportunities.

4.2.2.3. Refactoring and its prioritization. Once the feature is released, there is ATD left in the system. The ideal case is that the ATD would be completely removed by the system. However, this is not done or even possible according to our data, for two main reasons: part of the ATD is currently not known (see *uncertainty of impact* in Section 4.1.5), and the refactoring is usually only partially prioritized.

Prioritization of the refactoring depends usually on the kind of refactoring: the refactoring needed for easing (or especially allowing) the *short-term* release of features is usually prioritized and performed by the team. This is possible both because of the immediate clear business need of it and because such ATD can be refactored by being included in the successive feature development. Examples of such short term improvements include small or local (not spread out in the whole system) adjustments of patterns to allow a feature to be implemented. These characteristics are represented in the graph by the steep slope in the curve in correspondence to *short term effects*. As for the *long term effects* refactoring, usually it is represented by some extensibility or maintainability mechanism at a higher level of abstraction that has not been implemented during the development of the features. To introduce such mechanism the needed time is usually substantial compared to the feature development time: for example, if the refactoring is estimated to be 2 months and a new feature is supposed to take the same amount of time to be developed, it would not make sense to include the refactoring into the feature as a user story. Also, such task would probably influence other parts of the software, which might cause interruptions on other teams' work. For such reasons, such (as explained in Section 4.1.1.4) ATD is usually low-prioritized in favor to feature release.

4.3. Comparison of refactoring strategies

We will show what implications, in terms of refactoring strategies, can be drawn from the current results. We show this by analyzing the combination of the models previously shown and how they lead to different outcomes (Figs. 5 and 8), with respect to different refactoring strategies. In particular, we show how projecting several instances of the *phase model* over time and combining such projections with two different refactoring strategies, *refactoring maximization* and *refactoring minimization*, would bring different outcomes in term of development crises.

Refactoring maximization in Fig. 5 is represented the scenario in which both short-term and long-term ATD are refactored. We can see how the constant ATD and the delay effect from the *unknown* ATD are continuously accumulated, making the accumulation

strictly monotone even when the refactoring is maximized. This would exclude the possibility, for the companies, to apply a *complete refactoring* strategy, in which all the ATD is removed. The maximization of refactoring consists of, for the company, adopting a *partial refactoring* strategy.

Refactoring minimization in Fig. 6 we show the case in which a *no refactoring* strategy is applied in contrast with the previously mentioned *partial refactoring*. Even though the feature might be released earlier with respect to *partial refactoring* (when all possible refactoring was performed), the delay caused by the ATD grows because of the addition of *unknown* and *short-term* ATD.

Short-term comparison of strategies in Fig. 7 we have represented a further projection of both strategies for a hypothetical number of 3 releases (the same effect might be reached with a different number of releases, depending on the context): in the *no refactoring* strategy we can see the increasing of the delay caused by the ATD with *long-term effect* with respect to the *partial refactoring* function. In conclusion, after some time *partial refactoring* might be as convenient (in terms of features released) as *no refactoring*, with the difference that in the second case there is more ATD in the system (Fig. 8).

Long-term comparison of strategies By projecting the same trend for longer time and combining it with the *crisis point* model (assuming that the same crisis will happen when the same amount of ATD is accumulated), we can see that with a *partial refactoring* strategy the crisis point would be delayed (Fig. 9).

Choice of the optimal refactoring strategy with respect to the crisis points: taking in consideration the crisis points perspective, the main choice for the companies is to balance the prioritization of refactoring in order to have as less number of crisis points as possible, or to delay the crisis point over time according to the lifecycle of the product. The evidences collected suggest that the best strategy to avoid development crises is the partial refactoring.

4.4. Detailed case-study

We conducted a follow-up, in-depth case-study with one of the cases involved previously in the investigation. Specifically, we studied the case C_2. We investigated one of the recent refactorings conducted at the site in order to find patterns that would evaluate our previous hypotheses (as recommended by Yin and Runeson and Höst [20,22]).

Since we wanted to show the relationships among the factors over time, we will describe the case in a chronological fashion. We highlight key events in the timeline and then we describe what happened between the two events. We will focus only on the part of the system that was studied during the interviews.

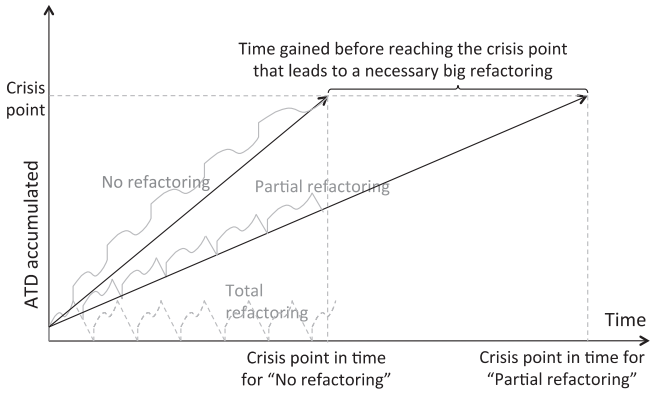

Fig. 9. Long-term comparison of the strategies: given the trend of inevitable accumulation of ATD, a crisis point will eventually be reached in both cases. The gain for the company is to reduce the number of crises and therefore the number of costly refactorings.

4.4.1. Chronological narrative of events

T_0: start of development: a new product of the product-line was requested by the market and by specific customers. Such product would include a set of features, but would also need to be integrated with the previous platform. Within the product, one of the components (that we call $Comp_A$) was integrated in the platform since it contained functionalities shared by to other components (that we call $Comp_{1\ldots n}$) included in different applications. $Comp_A$, being a central component, had (legal) dependencies with the other components, as visible in Fig. 10.

$T_{RefCompA}$: discover of the need of refactoring. After the integration of $Comp_A$ and after the component had been in use within the product for around 3 years, the system responsible found that there was an anomaly in the number of defects reported in the system, which led to a crisis. After the analysis of the bug reports, it was possible to understand that the design of $Comp_A$ was the cause for such anomaly (many problems were tracked back to $Comp_A$). Given the situation, the system architect and the managers decided to prioritize the refactoring of $Comp_A$. Such refactoring involved the restructure of $Comp_A$, considered not well modularized and too complex. The estimation for $Comp_A$ was considered as RefA. $Comp_A$ was therefore duplicated (branched) in $Comp_A'$, in order to be refactored in parallel to the continuous development of features. It was not possible, obviously, to stop the use of the product(s) using $Comp_A$ and its development during the refactoring, since the overall re-design of $Comp_A$ required a substantial amount of time (around 6 months).

$T_{RefComp1\ldots k}$: during the estimation for Ref_A, however, the impact of the changes implemented in $Comp_A$ with respect to $Comp_{1\ldots n}$ was not correctly understood and estimated. After the refactoring of $Comp_A'$ started, it became clear that the APIs needed to be updated, and the work needed to refactor $Comp_{1\ldots n}$ according to the changes in $Comp_A'$ required more time than expected. The refactoring of $Comp_{1\ldots n}$ was therefore estimated and broken down to stories that would need to be prioritized in the backlog of the teams allocated to such work. However, not all the refactoring work could start in all $Comp_{1\ldots n}$ before the changes in $Comp_A'$ would be completed. Therefore, the refactoring was initiated in $Comp_{1\ldots k}$ but not in $Comp_{k+1\ldots n}$.

T_{NewReq}: New application requirements involving $Comp_{k+1} \ldots Comp_n$. While the refactoring of $Comp_A'$ was still on-going, new feature requirements were received by the teams developing $Comp_{k+1\ldots n}$. The backlogs were re-prioritized and the new stories were given a priority higher than the priority of the refactoring stories. Such decision was also due to the fact that $Comp_A'$ was not ready at such moment. However, the stories for coordinating the refactoring with $Comp_A'$ remained low-prioritized also when it became possible to start.

$T_{NewComp}$: New applications. Starting during this time, new features were requested by the market and the customers. Therefore,

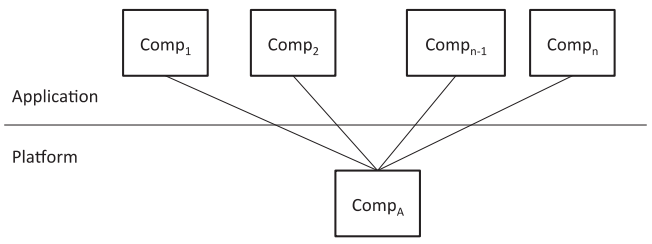

Fig. 10. Simplified architecture layers (Platform- and Application-layer) of the studied system.

new application components were created ($Comp_{n+1\ldots m}$). Such components were designed to interact with $Comp_A'$, since such component was being refactored and was considered more updated and better usable by the new application due to its re-design.

$T_{NoRefCompk+1\ldots n}$: When the refactoring of $Comp_A'$ was completed, the refactoring of $Comp_{k+1\ldots n}$ were still not carried out. Despite the fact that the applications were using the old $Comp_A$, such motivation was not enough to high-prioritize the stories for refactoring. From this point on, the system was using a duplicated component ($Comp_A$ and $Comp_A'$), in which part of the applications were interacting with $Comp_A$ and part of them were interacting with $Comp_A'$, causing a duplication of maintenance work.

4.5. Evaluation

To evaluate the exploratory results collected in phase I and II, we conducted and analyzed interviews (phase III) and the in-depth case-study (phase IV). In this section we show how the factors and the models where evaluated by the two approaches.

4.5.1. Factors evaluation

We deductively analyzed the case-study in order to recognize the factors previously obtained from the multiple case study. Fig. 11 shows a time-line with the major events explained in the previous section. Part of the events cannot be referred to a single point in time, but rather to a time interval between two time points. We mapped the factors to the time-line, in order to visualize their relationships over time (used in the next section for visualizing pattern matching). The following factors were recognized:

- **Uncertainty of use case in early stages:** citing the interviewee: "*the team did not completely understand the impact of the developed component on the users of such component*". Such factor was occurring at least in the beginning when the ATD was accumulated, but also in the beginning of the refactoring, where not all the users (stakeholder) of the refactored $Comp_A$ were identified. For example, the interviewee mentioned the lack of awareness about part of the testers interacting with $Comp_A$.
- **Uncertainty of impact:** citing the system architect: "*at time [$T_{RefCompA}$] it was easy to think that the issues [ATD] could be solved by just refactoring [$Comp_A$]*". Clearly the team did not understand the impact of introducing ATD into the system, affecting $Comp_{1\ldots n}$ and therefore ignoring the impact of changes.
- **Parallel development:** this factor is clearly present and influencing the accumulation of ATD for the whole period. First of all, the product was developed in parallel to the other products, which contributed to "isolate" the development from the users of $Comp_A$ and the development of $Comp_{1\ldots n}$. Such isolation contributed to the *uncertainty*.
  After the refactoring of $Comp_A$ was started at $T_{RefCompA}$, the parallel backlogs caused a misalignment in the prioritization of the distributed stories related to completing the refactoring, which led to the duplication of component $Comp_A$ (accumulation of ATD through uncompleted refactoring).
- **Time pressure:** the need for delivering quickly (short lead time) was mentioned especially in the beginning before the reaching of the crisis point $T_{RefCompA}$, and during T_{NewReq}, after the new requirements came in and new stories were added to the distributed backlogs. In the middle of the analyzed time-span, the refactoring was started, so it could be easy to think that the time pressure decreased. However, the reaching of the crisis triggered the refactoring, which was started in order to avoid the time spent on fixing a large amount of bugs instead of developing new features. We can see then how the time pressure played a role in that period as well.
- **New business requirements:** new requirements were obviously received at T_0, when the product development started, and at T_{NewReq}, when new features and customer-specific products were requested or identified by the product owners.
- **Priority of features over product:** the low-prioritization, during T_{NewReq}, of the stories related to complete the refactoring is a clear sign of this factor, which directly led to the uncompleted refactoring and the duplicated component (ATD).
- **Uncompleted refactoring:** from the start of the refactoring of $Comp_A$, the ATD accumulated through the duplication of the component was constant. However, the ATD (duplication) was not considered as such (but only temporary duplication) until it became clear that the substitution of $Comp_A$ with $Comp_{A'}$ was not possible due to the low-prioritization.

In conclusion, the case study confirmed the presence of many of the factors listed in Section 4.1 that influenced the accumulation of ATD over time.

4.5.2. Models evaluation

For each model presented in Section 4.2, we show how the evaluation interviews in phase III and the case-study conducted in phase IV brought evidences to the exploratory results of phase I and II. In particular, the distribution of the factors in Fig. 11 should be compared with Figs. 3 and 4 in order to understand the matching of the patterns.

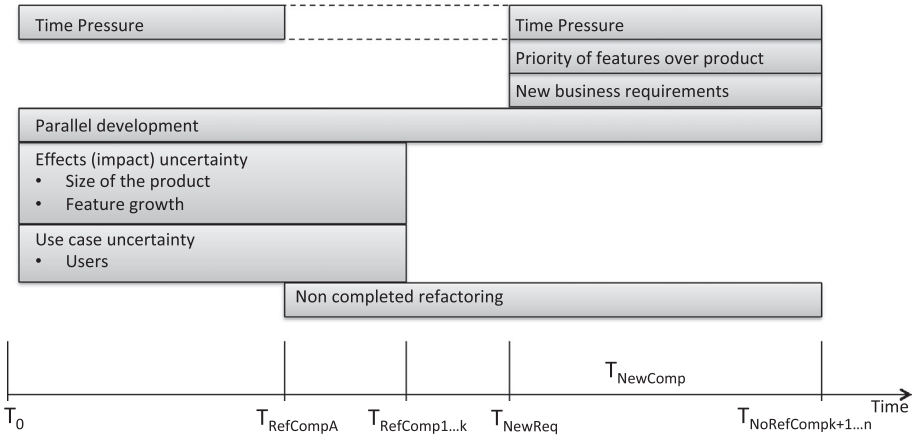

Fig. 11. Timeline of the case-study in relation with the factors influencing the accumulation of ATD over time.

4.5.2.1. Crisis-based ATD management (Section 4.2.1)). During the evaluation interviews we collected data on the models from company A, B, and the three cases $C_{1,2,3}$. We asked if the crises occurred recently and how the informant experienced them. We list a selection of relevant quotations from different informants all from different companies (not specified for confidentiality reasons). We cannot report all the evaluation data due to space constraints, but the ones listed here represent a good sample of the data used for models evaluation.

Chief architect:

"*When we realized that we were going into a situation where we had so many variants that the monolithic structure wouldn't be able to sustain that feature growth and those types of variance. We actually realized this before we hit the crisis point. It wasn't that we weren't able to deliver projects. We realized it before. So we didn't actually hit the crisis point and people stopped delivering software, but we were pretty close. We could see that the next project will not be able to deliver. So this will be a more expensive project. We have the luck that we got buy in from management that we needed to do these changes.*"

System architect:

"*So we had to take resources from feature development to stabilization. We just stopped feature development to stabilize what we had first. Then we could continue with features.*"

Software architect:

"*I think we have exactly the situation you described [referred to the slide showing the description of crisis point].*"

System architect:

"*[] we had to stop introducing new features and just fixing bugs. And then we have also started a redesign of some components. [. . .] And then the management understood that we need to do something about this component, redesign [. . .]*"

System architect:

"*I recognize [. . .], we had a lot of [bug fixing] in the same area. And we couldn't continual fixing here and there with sort of workarounds or not clean fixing. So we realized we needed real refactoring to solve all together and to ease the introduction of new features.*"

From the 5 companies interviewed, it is quite clear that the informants agreed on the validity of the model and they shared several experiences.

We can also see confirmatory evidences that the crisis model is valid for the in-depth studied case (Fig. 11). The crisis when the ATD was revealed is represented by the high number of defects, which led to allocate a large percentage of development to bug fixing. Such activity decreased the time dedicated to new features, together with the perception of the slowing down of development feature themselves.

4.5.2.2. Constant ATD accumulation (Section 4.2.2.1). In the studied case we can see how it was never possible to avoid the presence of ATD in the system, even by prioritizing the big refactoring. We could evaluate most of the factors that we have claimed to be the cause of constant accumulation of ATD. In the case study, *parallel development* is easily visible as influencing factor for the whole time, while *business evolution* is also quite persistent, excluding the phase after the crisis, when the impossibility to develop led to the refactoring.

4.5.2.3. Phases of ATD accumulation (Section 4.2.2.2). We can compare the model of the phases for accumulation and refactoring of ATD (Fig. 3) with the distribution of the factors in Fig. 11 related to the case-study time-line, in order to understand the matching of the patterns.

Matching patterns:

- *Uncertainty of use cases* in early stages is confirmed as a factor boosting the accumulation of ATD.
- The case suggests how the *uncertainty of impact* would also be extended until the actual refactoring was started.
- *Parallel development* was impacting the accumulation of ATD. The news is, parallel development had also an impact on refactoring
- The "inflow" of new business requirements (*business evolution*) influenced the accumulation both in the beginning of the development and also when new features were added.
- Before the second wave of requirements, there was a time span when the refactoring was started. However, we cannot map this decrement of the ATD with the *refactoring opportunity* hypothesized in Fig. 4, since the main reason for such refactoring was not the lack of time pressure, but rather the need to fix the bugs (and the source of them).
- *Time pressure* is present throughout the whole time in the case-study. Although the refactoring has been prioritized, such decision was taken because the current development was so slowed down that it was too difficult to add new features. The refactoring was therefore also prioritized in order to improve lead time (*time pressure*). Therefore, time pressure is to be considered as a constant accumulator for ATD. *Urgency,* in Fig. 4, was present close to the release, but it is just part of the overall time pressure, which is spread throughout the whole process. In order to decrease the lead time, management focuses on different target in a reactive way: first development, then refactoring when development starts to become difficult (approaching of the crisis).

In conclusion, most of the accumulation phases and components of the model have been confirmed, even though their spanning might vary and some components could not be confirmed by this specific case study: uncertainty is more present in the early stages but might be quite protracted, time pressure is constant throughout the development, as well as parallel development and business evolution. The *refactoring opportunity* (visible in Fig. 4) needs to be further confirmed: in the specific case study, the refactoring was high-prioritized for the same reasons why it was low-prioritized in other phases, i.e. for time pressure.

4.5.2.4. Refactoring prioritization strategies (Sections 4.2.2.3 and 4.3). The case-study is a good example of how the long-term refactoring was prioritized. Only after a crisis the refactoring was reactively started, exactly as shown in Fig. 3. We cannot compare this chain of events with an hypothetical case where the long-term refactoring would have been prioritized in advance, but we can see how the project reached a crisis, and from the issues reported it was possible to identify the presence of long-term ATD as a source of the crisis. We can therefore infer, from the evidence, that the *No refactoring* strategy led to the crisis, but we do not know how faster than if the debt would have not been taken or refactored earlier. Once there, postponing architectural refactoring (long-term) resulted quite costly and was not completed, leading to accumulation of different ATD. Therefore, the strategy of postponing long-term ATD refactoring after the crisis seems to be a non-optimal solution. We can consider these data as partially confirming the previous models, considering though the need for further evidences from more cases in this direction.

5. Discussion

We have conducted an exploratory multiple-case study in 7 large software organizations, showing factors causing the accumulation of ATD (RQ1), trends in such accumulation and refactoring over time (RQ2) and analyzing different outcomes (in terms of development crises) for different refactoring strategies (RQ3). We have backed up the exploratory results, collected in 2 phases, with cross-company evaluation interviews and an in-depth case study about a concrete ATD case, which strengthened the results with several confirmatory evidences.

The provided qualitative representation shows factors and trends that reveal, in our opinion, important implications in the light of the recently emerging practice of ATD management. One very important variable to be taken into consideration is time, and we have done a first step in order to explain the relationship between such impacting variable and the phenomena of ATD accumulation and refactoring. In the followings we therefore discuss a number of implications that can be inferred by our results. They represent hypotheses qualitatively tested through a substantial number of experts from similar domains or through an in-depth case-study, but that would certainly benefit from being quantitatively complemented and further assessed in the future.

5.1. Implications for research

The investigation in this paper brought to light several results throughout different phases, especially during the exploratory phases I and II. Some of such results were evaluated through phases III and IV, where we run interviews and we conducted an additional, evaluative case study. We add an evaluation table (Table 2), were we show a summary of the results and by which means they have been evaluated.

These results represents a first step towards the development of middle-range theories [24], which are not considered universally valid, but only within a range of contexts. We investigated the results in 7 similar sites, which allows us to claim the validity of our findings to the context of large software companies developing embedded software and employing ASD. Another attribute of the context is the geographical location of the companies, all placed in Scandinavia. In different geographical areas the results might be different. We therefore encourage further investigation of the results in other contexts in order to make another step towards further generalizing the results (as explained in the external validity section).

Answering the RQs contributes to the body of knowledge, according to the gaps identified in a recent systematic mapping study [4], by providing a reliable industrial study based on the experiences of several ATD stakeholders: architects, developers and product owners. In particular:

- RQ1: we have studied the socio-technical factors related to phenomena of accumulation and refactoring of ATD. These factors might be studied and treated separately. Such factors offer a better understanding of the overall phenomenon of ATD accumulation and refactoring, which is a prerequisite for its management.
- RQ2: we have developed and evaluated two qualitative models of the trends in accumulation and refactoring of ATD over time based on its prioritization: the crisis model and the phase model.
- RQ3: we have provided constraints and recommendations about different refactoring strategies and their effects on development crises. Such recommendations should be tested with further case studies.

5.1.1. ATD and software architecture management

As we can see from Fig. 4 there is a constant accumulation of ATD for several reasons, some of which are also external to the company. In conjunction with this, part of the ATD remains unknown. These two factors together lead to the consequence that each iteration brings a quantity of ATD in the system, and that part of it will remain. Even if the magnitude of such accumulation is not yet clear, the function over time is monotone. These results show two main findings important for the software architecture community: that it is quite likely that the initial software architecture would change because of changing requirements and that a number of sub-optimal solutions will always been implemented because of the several factors causing ATD. Therefore, such drifting needs to be managed continuously, especially to uncover the ATD

Table 2
Exploratory and evaluated results.

Results	Confirmatory evidences from multiple interviews	Confirmatory evidence from case-study	Exploratory evidence from multiple interviews	Exploratory evidence from case-study
Factors influencing ATD accumulation (RQ1)				
• Uncertainty of use-case in early stages	X	X		
• Business evolution	X	X		
• Time pressure	X	X		
• Priority of features over product	X	X		
• Split of budget	X			
• Design and Architecture documentation	X	X		
• Reuse of Legacy/third party/open source components	X			
• Parallel development	X	X		
• Uncertainty of impacts	X	X		
• Non-completed refactoring	X	X		
• Technology evolution	X			
• Lack of knowledge	X	X		
Models of ATD accumulation and refactoring (RQ2)				
• Crisis-based ATD management	X	X		
• Phases of ATD accumulation	(X)	X	X	
Refactoring strategies (RQ3)				
• *Partial refactoring* is the best option for the maximization of refactoring. *Complete is refactoring* not realistic.	(X)		X	X
• Drastic minimization of refactorings (*No refactoring*) leads to development crises often in the long run.	(X)		X	X

that it is unknown and in order to prioritize the ATD that have the worse effects over time. It is important to understand how to continuously analyze the architecture in order to develop better solutions and apply a suitable refactoring strategy.

5.1.2. ATD and Agile Software Development

The employment of Agile Software Development seems to bring both advantages and disadvantages to the phenomenon of ATD accumulation. Such influences are related to the incentives and disincentives previously mentioned. For example, the Agile process and principles favor the focus on the features over the product and boosts the accumulation of ATD, relying on a mandatory subsequent refactoring that is not always recognized from the management point of view. On the other hand, Agile provides an iterative process for gradually taking care of uncertainty and would allow to iteratively keep track of the ATD. Another trend related to ASD is having teams that have to focus on a feature and are free to touch any component. The lack of domain knowledge, however, boosts the accumulation of ATD.

From this investigation we can see how the chosen strategy used for ATD prioritization and management would have an impact on ASD: frequent crisis points might nullify the responsiveness and continuous delivery achieved by the Agile practices. Therefore, a set of lightweight practices for ATD management is needed and would benefit from the Agile iterative process if well embedded. We are currently studying such practices by employing action research at the companies involved in this study. The results contribute to the current body of knowledge according to [4].

5.2. Implications for practice

The results have implications for multiple roles in the organizations: from product managers and product owners, to architects and developers.

5.2.1. Implications for product managers and product owners

The goal for software companies is to avoid development crises, since platform creation and/or huge architectural refactorings are long and costly activities and would stop the continuous delivery of features to the customer. We have analyzed different refactoring strategies with respect to the minimization of crises. The evidence suggest that a *complete refactoring* strategy is not possible, and therefore that the main goal for a software company cannot be to have an "eternal" system, but rather to reduce the number of times the development crises are repeated over time before the retirement of the product. This means that a company needs to proactively adopt a partial refactoring strategy and needs to plan refactoring activities as part of the product development.

Some of the factors have a disincentive (or incentive) effect on ATD accumulation. The known disincentives are the ones described in Section 4.1.1 and related to business factors, such as having too much focus on features with respect to the product, having a split budget for project and maintenance and having high penalties on deadlines. To our current knowledge, such disincentives do not influence some specific ATD issues. However, future studies could increase the understanding on such matter. An important incentive, especially relevant for product owners, is the prioritization of ATD refactorings, especially the long-term ones.

5.2.2. Implications for architects and developers

In many cases, the model in Fig. 3 has been found valid to describe the current events. Automatic static analysis tools together with the current practices of lightly documenting software architecture do not seem to be able to provide enough awareness of the ATD present in the system, if not on an intuitive level. Therefore, more continuous analysis needs to be done in order to assess the current ATD in the system and its impact (interest). *Uncertainty of impact* might be decreased by identifying the debt, localizing it and understanding the stakeholders involved in the payment of the interest. Such practice might be supported by various metrics (usually context dependent), but the developers, architects and the involved stakeholder need to work together for the aggregation of information and the communication of the risk incurred in taking debt that has a wide impact on the organization.

5.3. Limitations

The graphs in this paper (Figs. 5, 6, 7, 9 and 8) are not meant to represent precise data coming from a measurement system. Therefore the steepness of the curves and the projections might vary in real context. The magnitude for the contribution of each factor is also to be further assessed. However, we offer the recognition of factors that are not necessarily possible to be measured and therefore discovered by quantitative analysis, such as urgency and uncertainty, and their relationship with time. The results are qualitatively developed through a thorough research process including several triangulation techniques recommended [20] for qualitative studies: we used a wide amount of qualitative data coming from many informants throughout four iterations of investigation (see Table 1), from seven sites and with different roles, which allowed us to compare and test statements among themselves. Furthermore, architecture documentation and improvement backlogs have been evaluated as secondary data. This has allowed us to apply source triangulation. We have also applied three ways of qualitative evaluation of the results: plenary workshops, interviews and a company-specific case-study. Table 2 shows which results have been consolidated with which method, and which are, on the other hand, still exploratory and need further evaluation. Another limitation is the use of a single, company-specific case-study: due to resource constraints, the authors could not perform the same in-depth investigation in other organizations, but we are confident that the whole research community, especially the empirical one, will contribute to this topic with more evidence in order to build an even more solid body of knowledge on the ATD management.

5.4. Future work

5.4.1. Specific interest of ATD items

The model of accumulation of ATD with respect to the crisis point does not separate the ATD from its interest. It was not possible to do such separation from the data analyzed during the current investigation. In order to further develop the models shown here, it is of utmost important to understand the interest associated with specific ATD items, in order to prioritize them and to understand which one contributes more to the total accumulation of extra-effort leading to a crisis point. An initial step has been done by the same authors of this paper in a recent paper accepted for publication [7].

5.4.2. Other socio-technical patterns related to ATD accumulation and refactoring

As we have highlighted in the case study, ATD is not only a technical problem, but involves several factors, from the psychological one to the organizational structure, to the processes. Understanding socio-technical patterns that favor (or limit) how ATD is accumulated might be useful for the proactive avoidance of its accumulation.

5.4.3. Further evaluation

The study of the ATD is quite recent and needs further evidences. We have provided multiple sources of evidences for some results, but for other, of more exploratory nature, the scientific

community needs further evaluation. We have compiled Table 2, which shows the results that need more evidences. Such results should not be considered unreliable, since they are based on the combined experiences of several employees from 7 sites in 5 large software companies. However, more in depth studies might be done in order to define more precise models and compare with multiple sources of evidence. We have done a first step in such direction.

5.5. Threats to validity

We discuss the threats to validity with respect to the guidelines proposed in [20]: construct, internal, external validity and reliability.

5.5.1. Construct validity

We did not use the ATD terminology (*debt*, *principal*, *interest*) during the investigation, since it could have been interpreted differently in different contexts. In order to keep construct validity, we operationalized the ATD as "architecture inconsistencies" (or alternatively "sub-optimal solutions" or in some cases "violations", depending on the class of ATD) with respect to the current desired architecture related to a specific case. With the same approach we operationalized the *principal* as the refactoring cost to obtain an optimal solution and the *interest* as the extra-effort caused by the ATD or other kinds of extra-impact. We investigated the actual existing cases with software and system architects, obtaining the best knowledge about both the desired architectures and a good explanation of the sub-optimal solutions in place. As for the principal and interest, we interviewed the developers involved in the case studied, in order to be sure that estimations and evaluation of effort were as accurate as possible.

5.5.2. Internal validity

Rather than investigating a direct cause-effect relationship, we have collected *architecture explanations* [24] from several cases, in order to understand which factors caused the accumulation of ATD and how such accumulation is handled in the specific cases. Since we aimed at understanding the causes of ATD accumulation, there exists a threat to internal validity. We need to take in consideration the possibility that other factors than the ones listed here would cause ATD, and that crises would be reached because of other projects' issues. We mitigated this threat by obtaining the information about the causes of the same ATD item from several roles, and we asked follow-up questions in order to probe the explanations. Also, collecting similar evidences supporting the same explanation across the cases contributed to strengthen our conclusions.

5.5.3. External validity

One of the major threats for case-study research is the ability to generalize from the case-specific results to other cases. Generalizing to a universal theory is not necessarily the goal for an engineering discipline: according to [24], middle-range theories, valid to a restricted ranges of contexts, result more useful in practice. In order to develop such middle-range theories, we have employed an analytical induction strategy to generalize from case-studies [24]. We have collected architectural explanations from contexts that are architecturally similar among themselves, but contain some differences (all of them are large companies developing embedded software and having similar organizations). The similarities allow the researchers to make the claims more robust (for example, as shown in the picture (Fig. 2), the same code was mentioned in three of the 7 cases), while the differences allow the extension of the findings, if similar, to such contexts as well. In our case, we have identified the findings that were confirmed by multiple sources (for example, the model concerning the crisis point was found repeatedly valid from all the sources) and the ones that need further investigation, in Table 2. As for the second kind of results, we do not say that such findings are not valid in all the contexts, but rather that it was not possible, with our data, to find strong confirmatory results. The table can be used to understand how much our results can be generalized based on the kinds of evidences obtained from the different sources. Although we do not claim the results universally general, we can say that for RQ1 and RQ2 the findings are supported by confirmatory findings crossing multiple contexts, and therefore can be considered reaching a generalizability of middle-range theories, where the range is represented by large Scandinavian software companies developing embedded software and employing ASD. As for RQ3, we report propositions that need to be further evaluated with different means than the ones employed in this study.

5.5.4. Reliability

As described in the methodology section, three authors participated in defining the research design, the questions and into checking the findings from the coding activities. The results were presented in a workshop at the end of each of the four phases to the industrial contacts involved in the investigation, in order to collect feedback and strengthen the reliability of the results.

5.6. Related work

Lehman et al. [25] propose a formal approach for process modeling. The paper emphasizes the usefulness of formal models (e.g. functions) for effort prediction. We have developed the crisis model (Fig. 3), which can be considered the abstract model, and we have done a first calibration by finding the factors that are needed to describe the formal function (as parameters of the function). Our approach can be considered as a first necessary step towards the formalization and the precise prediction of the process, which needs more quantitative data. An empirical model of debt and interest is described by Nugroho et al. [14]. However, such method only focuses on the interest paid during maintenance and it is not focused on finding the causes for the accumulation of debt. The business factors that we have found are missing in the study of ATD as also reported from a single case study [11]. Sindhgatta et al. [26] have studied the software evolution in an Agile project, where some of the Lehman laws were tested through project sprints. Some of the results suggest a confirmation of the trends that we have identified. For example, the laws of (continuous) change and growth show the monotonicity of system growth and the necessity for the system to adapt to the business environment, which are recognized also in our factors. However, the results are not directly connected with ATD.

6. Conclusions

Decisions on short-term and long-term prioritization of architecture refactoring need to be balanced and need to rely on the knowledge of the underlying phenomenon of ATD. The current management of ATD is an under-researched topic and we contribute to the empirical software engineering body of knowledge by reporting from a multiple case study investigating practitioners' experiences from 7 large Scandinavian companies employing Agile and developing product lines of embedded software.

In this paper we have shown what are the causes of the accumulation of ATD, and we outline, through the recognition of different influencing factors, clear objectives that can be treated or further studied in order to avoid or mitigate the accumulation of ATD (RQ1), therefore easing the ATD management for architects and managers.

We have also presented 2 models for describing the accumulation and refactoring of ATD over time (RQ2). Such models are the *Crisis Model* and the *Phases Model*. Such models can be further studied and tested with the conduction of experiments and the collection of quantitative data by the ISERN community.

Based on the models, we have identified possible strategies for refactoring ATD (RQ3) and we provided recommendations with respect to the minimization of development crises. We conclude that *complete refactoring* is not a possible strategy in the current studied companies, due to the continuous and inevitable accumulation of ATD and the impossibility of removing it all. The *No refactoring* strategy leads to crises points often, hindering the long-term responsiveness in providing new customer value, as required in ASD. The best strategy is therefore to apply *partial refactoring* to minimize crises and to push the crisis point as far as possible with respect to the lifecycle of the products. The results highlight different outcomes related to different ATD prioritization strategies, which would help architects and managers in balancing the ATD management strategies with respect to the business goals and the life-cycle of the products.

An important goal in research and industry is to improve the practices and tools to uncover ATD present in the system and to keep track of it. It is also important to identify the best points in time for performing refactoring and therefore repaying the debt that is going to generate more interest effort later on. Such practices need to complement the current Agile process in place, in order to keep responsiveness stable through the whole software development process.

Acknowledgments

We thank the companies that are partners of the Software Center, especially the software and system architects participating in the project run by the research team composed by the authors of this paper and led by the first author. We are also extremely grateful for the work done by Lars Pareto, member of the team and participating in the project until 2013.

References

[1] T. Dingsøyr, S. Nerur, V. Balijepally, N.B. Moe, A decade of agile methodologies: towards explaining agile software development, J. Syst. Softw. 85 (6) (2012) 1213–1221.
[2] P. Kruchten, R.L. Nord, I. Ozkaya, Technical debt: from metaphor to theory and practice, IEEE Softw. 29 (6) (2012) 18–21.
[3] E. Tom, A. Aurum, R. Vidgen, An exploration of technical debt, J. Syst. Softw. 86 (6) (2013) 1498–1516.
[4] Z. Li, P. Avgeriou, P. Liang, A systematic mapping study on technical debt and its management, J. Syst. Softw. 101 (2015) 193–220.
[5] M. Daneva, E. van der Veen, C. Amrit, S. Ghaisas, K. Sikkel, R. Kumar, N. Ajmeri, U. Ramteerthkar, R. Wieringa, Agile requirements prioritization in large-scale outsourced system projects: an empirical study, J. Syst. Softw. 86 (5) (2013) 1333–1353.
[7] A. Martini, J. Bosch, The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles, WICSA 2015, Montreal, Canada, in press.
[8] R.L. Nord, I. Ozkaya, P. Kruchten, M. Gonzalez-Rojas, In search of a metric for managing architectural technical debt, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012, pp. 91–100.
[9] W. Cunningham, The WyCash portfolio management system, ACM SIGPLAN OOPS Messenger 4 (1992) 29–30.
[10] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al., Managing technical debt in software-reliant systems, in Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, 2010, pp. 47–52.
[11] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A.L.M. Santos, C. Siebra, Tracking technical debt—an exploratory case study, in: 27th IEEE International Conference on Software Maintenance (ICSM), 2011, 2011, pp. 528–531.
[12] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, A. Vetro, Using technical debt data in decision making: Potential decision approaches, in: 2012 Third International Workshop on Managing Technical Debt (MTD), 2012, pp. 45–48.
[13] J.-L. Letouzey, The SQALE method for evaluating technical debt, in: Proceedings of the Third International Workshop on Managing Technical Debt, Piscataway, NJ, USA, 2012, pp. 31–36.
[14] A. Nugroho, J. Visser, T. Kuipers, An empirical model of technical debt and interest, in: Proceedings of the 2nd Workshop on Managing Technical Debt, New York, NY, USA, 2011, pp. 1–8.
[15] K. Schmid, A formal approach to technical debt decision making, in: Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures, 2013, pp. 153–162.
[16] ISO – International Organization for Standardization, System and software quality models <http://www.iso.org/iso/catalogue_detail.htm?csnumber=52075> (accessed 08.03.15).
[17] P. Kruchten, What do software architects really do?, J Syst. Softw. 81 (12) (2008) 2413–2416.
[18] A. Martini, L. Pareto, J. Bosch, Role of architects in agile organizations, in: J. Bosch (Ed.), Continuous Software Engineering, Springer International Publishing, 2014, pp. 39–50.
[19] M.A. Babar, I. Gorton, Comparison of scenario-based software architecture evaluation methods, in: 11th Asia–Pacific Software Engineering Conference, 2004, 2004, pp. 600–607.
[20] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empir. Softw. Eng. 14 (2) (Dec. 2008) 131–164.
[21] A. Strauss, J.M. Corbin, Grounded Theory in Practice, SAGE, 1997.
[22] R.K. Yin, Case Study Research: Design and Methods, SAGE, 2009.
[24] R. Wieringa, M. Daneva, Six strategies for generalizing software engineering theories, Sci. Comput. Program. 101 (Apr. 2015) 136–152.
[25] M.M. Lehman, G. Kahen, J.F. Ramil, Behavioural modelling of long-lived evolution processes: some issues and an example, J. Softw. Maint. 14 (5) (2002) 335–351.
[26] R. Sindhgatta, N.C. Narendra, B. Sengupta, Software evolution in agile development: a case study, in: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, New York, NY, USA, 2010, pp. 105–114.